Modification and Evaluation of Linux I/O Schedulers

Asad Naweed, Joe Di Natale, and Sarah J Andrabi
University of North Carolina at Chapel Hill

***Abstract*—In this paper we present three different Linux I/O schedulers—Average Queue Length, Read-Write FIFO and Scheduler Selector. We then present an performance analysis of these three schedulers, the Linux default CFQ scheduler and Random I/O scheduler based on different workloads and an analysis of the performance when multiple processes are running simultaneously.**

I. Introduction

Disk operations —specifically disk seeks —are one of the slowest operations on modern computers. Therefore, when the kernel gets an I/O request, it does not directly issue the block requests. Instead, it attempts to maximize performance for different kinds of workloads by using different scheduling mechanisms. The Linux 2.6.32 kernel comes with four I/O schedulers: no-op, deadline, anticipatory, and completely fair queuing (CFQ), with CFQ being the default. The kernel provides the option of switching the I/O scheduler either at boot time or run time depending on *a priori* knowledge of the workload. The I/O scheduler can be modified or extended, and the new scheduler can be added to the kernel as a loadable module. Alternatively, the kernel can be recompiled based on the modifications made to the scheduler.

In this paper, we present three new I/O schedulers that build on various concepts of existing I/O schedulers and borrow some ideas from the process schedulers. These schedulers include a Read-Write FIFO scheduler and an Average Queue Length (AQL) based I/O scheduler. Along with these three, we implement a random scheduler, which as the name implies randomly chooses requests from a list and schedules them. A scheduler selector is also implemented, which chooses from one of the schedulers present in the kernel during run time based on I/O performance. We report the performance of all five schedulers on a set of workloads, which represent a range of I/O behavior. We use various mixed workloads with several different types of I/O requests concurrently accessing the I/O subsystem. In such a scenario, it is expected that the I/O scheduler will not deprive any process of the required I/O resources. We analyze the throughput of the I/O schedulers and observe the behavior of the schedulers with respect to the cache size.

We ask the following questions for our analysis: What is the impact of extended schedulers on the execution time of some realistic benchmarks? Do the scheduling policies provide similar performance under different workloads, or is the performance different? If it is different, then how different? How does random scheduling perform as compared to the other schedulers? Is it better or worse? Do our schedulers provide any additional performance benefits? How is the performance of the schedulers affected by varying the cache size?

The paper is organized as follows. Section II briefly describes the working of the Linux kernel 2.6.32. In section III, we give an overview of the three schedulers we implemented. In section IV, we describe the working of the scheduler selector. In section V, we present our experimental evaluation criteria and test cases using benchmarks. Observations are presented in section VI, and the conclusion is presented in section VIII. Section VII describes the contribution of each of the team member.

II. Description of Linux I/O Schedulers

This section briefly describes three of the four schedulers provided by Linux 2.6.32, the no-op scheduler, deadline scheduler and the Completely Fair Queuing (CFQ) Scheduler. These schedulers are described here because our implementations of the Linux I/O Schedulers take ideas from these two schedulers.

A. CFQ Scheduler

The goal of CFQ is to equally distribute I/O bandwidth equally among processes performing I/O operations[1]. The CFQ scheduler utilizes a hash on the process identifier (PID) of the process wanting to queue an I/O request to place it in the appropriate queue. In Linux kernel version 2.6.32, the CFQ implementation utilizes 64 separate queues. During dispatch, requests are pulled from the head of each non-empty queue. The requests selected for dispatch are then sorted and merged into the dispatch queue for the device driver to handle.[3]

B. Deadline Scheduler

The goal of the deadline scheduler is to guarantee a maximum latency per request.[4] Read requests have lower deadlines because processes usually block waiting for read I/O to finish. The scheduler maintains a set of queues for both read and write requests. Both sets of queues consists of a FIFO queue based on deadline and a queue sorted in increasing Logical Block Address (LBA) order. At the end of a dispatch, the scheduler determines if write requests have been starved and if a new batch of reads or writes should be dispatched.[4] At the start of each dispatch, the FIFO queue is checked to determine if any deadlines are passed. If no deadlines have passed, then the scheduler pulls from the appropriate LBA sorted queue.[4]

C. No-op Scheduler

Noop scheduling is a simple FIFO based, low overhead scheduler.[1] Limited merging takes place and is done with a last hit cache.[2] This scheduler is normally utilized when the block device is fast (i.e. an solid state drive) and the system is CPU bound.[2]

III. Description of New I/O Schedulers

This section describes the I/O schedulers that we implemented, the Average Queue Length I/O Scheduler, the Read Write FIFO I/O Scheduler and the random I/O Scheduler. The schedulers described here built up on the concepts of the current Linux I/O schedulers as described in the previous section.

A. Average Queue Length I/O Scheduler

The primary goal of the AQL I/O scheduler is to dynamically scale the number of I/O requests satisfied based on the amount of I/O each process is requesting. When all processes have approximately equivalent need for I/O, then AQL acts similar to CFQ by attempting to give each process equivalent I/O bandwidth. When processes have differing I/O requirements, AQL will attempt to dispatch more requests for the processes that require more I/O. Processes will never starve because AQL will always dispatch at least one request from a busy queue. If the number of processes is high, AQL will behave similarly to CFQ. Requests are merged into the dispatch queue in sorted order to maximize block device performance.

AQL maintains a set of 64 queues and I/O requests are added to a queue based on the PID of the requesting process. The scheduler maintains a running average of the queue length for each queue which allows it to gauge the amount of demand for a set of processes. The average length for each queue is updated each time a dispatch occurs. If a queue is idle, its running average will be halved. If the queue is idle for 10 dispatch cycles, AQL assumes the process making I/O requests was either terminated or won't utilize I/O for some time in the future. In this case, the queue's running average is reset to 0. The number of requests dispatched from a queue is determined by the average length of the queue divided by the sum of the average lengths of all busy queues.

B. Read Write FIFO I/O Scheduler

The read-write FIFO scheduler maintains two separate lists, one for read requests and one for write requests, which are stored in a FIFO fashion

in the list. During the enqueue phase the requests are stored in one of the lists based on whether it is a read or a write request. Scheduling a request to a disk drive involves inserting it into a dispatch list, which is ordered by block number and deleting it from the list, for example the read fifo list. A set of contiguous requests is moved to the dispatch list. Requests are selected by the scheduler using the algorithm presented below. The algorithm treats 5 contiguous requests at a time.

Step 1: If there are read as well as write requests in the fifo lists, then select 60% reads and 40% writes and dispatch them and exit

Step 2: If there are only read requests and no write requests, then all the requests dispatched are read requests and exit

Step 3: If there are only write requests and no read requests, then dispatch all writes requests and exit

Step 4: If there are less than 60% read requests, then dispatch all the reads and the remaining dispatches will be write requests to ensure at least 5 contiguous requests are dispatched.

When the scheduler dispatches requests it gives preference to read requests over write requests. This policy prevents write starvation because no matter what a write request, if present, will always be dispatched. However in that case the write performance will be poorer but writes will not be starved.

C. Random I/O Scheduler

The Random I/O scheduler, as the name implies selects schedules I/O requests randomly. The scheduler maintains one list of I/O requests. During the enqueue phase each request is added to the tail of the list. Scheduling the request involves randomly selecting a request from the list and inserting it into a dispatch listthe request is added to the tail of the dispatch list. The dispatch queue in this case is not sorted by block number. Once the request has been added to the dispatch list it is deleted from the Random I/O list.

The Random I/O scheduler can starve certain requests, if it never chooses those request from the list. Since the way that random chooses requests from the list is random, there is no way to ensure that it chooses all the requests currently in the list before choosing a newly added request. This has an impact on the performance of the scheduler. A study of its performance under a range of workloads is presented in Section V.

IV. I/O Scheduler Selector

The IO scheduler selector does not implement its own scheduling algorithm. Rather, it discovers and utilizes other elevators already registered with the kernel and then switches between those schedulers based on a heuristic function that estimates throughput. We assume that each registered elevator has its own strengths and weaknesses, i.e. a different level of performance for different kinds of workloads. As the workload changes, so should the scheduling algorithm. At the high level, the procedure is relatively straightforward. We start with a randomly chosen elevator, and then begin monitoring throughput. If the throughput performance drops below a certain threshold, we switch to another random elevator. In our experiments, we found that our switching algorithm quickly converged to the most optimal elevator for the given workload at any point in time. The selector also adapts to changing workloads, and quickly selects the most optimal scheduler. Since the choice of elevator at each switch is completely random, the selector takes $O(n)$ time to converge in the worst case, where n is the number of registered elevators in the kernel. However, the expected and observed time was not always the worst case.

There were various engineering challenges associated with this implementation. According to the Linux elevator API, each scheduling algorithm has to have its own elevator_type and elevator_queue. Also, the kernel does not provide a mechanism for the discovery of various elevator types and obtaining handles to the elevator_op functions of

those elevators. We modified the kernel source code, exposing certain static functions, including elevator_find and elevator_get, to obtain this information, and also help in draining queues and perform elevator switching on-the-fly, without creating a new elevator_queue. The IO selector was registered as an elevator inside the kernel, and we could not use traditional methods of switching elevators, because that would mean un-registering the selector. The actual switching of the different elevators was done by means of a new kernel thread inside the selector, which continuously monitored throughput, and switched the elevators when necessary. Care had to be taken while draining queue prior to calling new function handlers, and also to ensure that incoming requests to the request_queue did not disrupt the switching process. This was done by placing the queue on "bypass", which indicated to the underlying IO scheduler inside the Linux kernel to stop using the elevator and perform a simple FIFO procedure on the request queue to dispatch requests.

The scheduler selector chooses from Deadline, Anticipatory, no-op and AQL.

V. Experimental Evaluation

This section presents the benchmark setup, the experimental platform as well as the different workloads used for the performance analysis of the Linux CFQ scheduler, the AQL I/O scheduler, the Read/Write FIFO scheduler, Random I/O scheduler and the Scheduler Selector. The results provided by the scheduler selector are a mix of the results of different schedulers as already explained in Section IV. The workloads are described in section V.D. For our analysis we use IOzone to perform benchmarking of all I/O schedulers.

A. Benchmark Setup

The IOzone benchmark tool [5] is a workload generator for a variety of file operations. IOzone provides a variety of file system performance coverage thus giving a broader idea of what the performance of the system will be like for different scenarios.

IOzone tests performance by using file operations to access files of different sizes and accessing different sized chunks of the file at a time. It calls these transfer size chunks record sizes. IOzone provides an automatic mode that produces output that covers all tested file operations for record sizes of 4k to 16M for file sizes from 64k to 512M. File creation time is not counted in the benchmark and we do not include time taken to close the file in the measurements. IOzone also allows us to report the CPU utilization for each of the five schedulers for each of the file operations that were run using automatic mode. Another set of tests allows to analyze the throughput for a certain number of processes/threads running on the machine. The test is configured to run 5 processes each of which accesses a 100MB file in 4KB records. Each test runs for a number of different file operations. The throughput tests are executed for each of the five schedulers on our platform. The average, minimum and maximum throughput obtained for each of the file operations is reported.

B. Experimental Platform

We conducted the following experiments on a single core 2.4 GHz Intel Xeon processor system, with 1GB main memory and 512MB L2 cache, running CentOS (Linux 2.6.32) and bus speed of 100MHz. Only a single processor is used in this study. For each experiment we reboot the machine to remove any cache effects.

C. Metrics

For the benchmark experiments the metric aggregate disk throughput (in KB/s) is used to demonstrate the performance of the schedulers. With no other processes executing in the system (except daemons), I/O intensive application execution time is inversely proportional to disk throughput[6]. In such situations, the scheduler with the largest throughput (smallest execution time) is the best scheduler for that file operation. The I/O schedulers that we test viz. CFQ, RWFIFO and random do not favor any particular process. The Average Queue Length I/O scheduler favors processes which have

TABLE I
I/O SCHEDULER THROUGHPUT FOR WRITES AND MIXED WORKLOAD WITH 5 PROCESSES

Scheduler	Initial write	Rewrite	Mixed workload	Random write
CFQ	89800.77	62254.98	596536.20	7343.94
Random	52507.09	26557.54	238565.50	6008.48
Average Queue	75470.37	67863.41	595586.67	7750.88
RWFIFO	85936.97	61289.21	576464.18	7333.17
Scheduler Selector	57647.29	52716.48	571131.52	6032.48

TABLE II
I/O SCHEDULER THROUGHPUT FOR READ OPERATIONS WITH 5 PROCESSES

Scheduler	Read	Re-read	Stride read	Random read
CFQ	1189923.14	1199803.94	827698.06	794624.25
Random	505763.85	508004.99	356562.33	324940.65
Average Queue	1077215.14	1195787.41	757119.36	789136.14
RWFIFO	1186422.03	1193834.84	824482.86	793497.72
Scheduler Selector	1126011.22	1138910.12	791094.59	766592.45

more I/O requests maintaining a running average of the number of I/O requests per process. None of these schedulers however, introduces any form of delay to favor one application over another and thus has no effect on the execution times of the applications run during the benchmark test. RW-FIFO gives preference to reads over writes and thus the performance comparison for these two types of operations, along with one for a mixed load is an important metric for this scheduler. An interesting thing to note will be the effect on performance due to the scheduler switching done on the fly by the scheduler selector.

D. Workload Descriptions

The following represent all the various workloads used for the two experiments using IOzone: **Write**: This test measures the performance of writing a new file. When a new file is written not only does the data need to be stored but also the overhead information for keeping track of where the data is located on the storage media—metadata.It is normal for the initial write performance to be lower than the performance of rewriting a file due to this overhead information [5].

Re-write: This test measures the performance of writing a file that already exists. When a file is written that already exists the work required is less as the metadata already exists [5].

Read: This test measures the performance of reading an existing file.

Re-Read: This test measures the performance of reading a file that was recently read. Performance in this case tends to be higher as the OS caches the data recently accessed.

Random Read: This test measures the performance of reading a file with accesses being made to random locations within the file.

Random Write: This test measures the performance of writing a file with accesses being made to random locations within the file.

Random Mix: This test measures the performance of reading and writing a file with accesses being made to random locations within the file.

The performance of a system under random reads, writes and mix type of activities can be impacted by several factors such as: Size of operating systems cache, number of disks, seek latencies, and others [5]

Stride Read: This test measures the performance of reading a file with a stride access behavior.

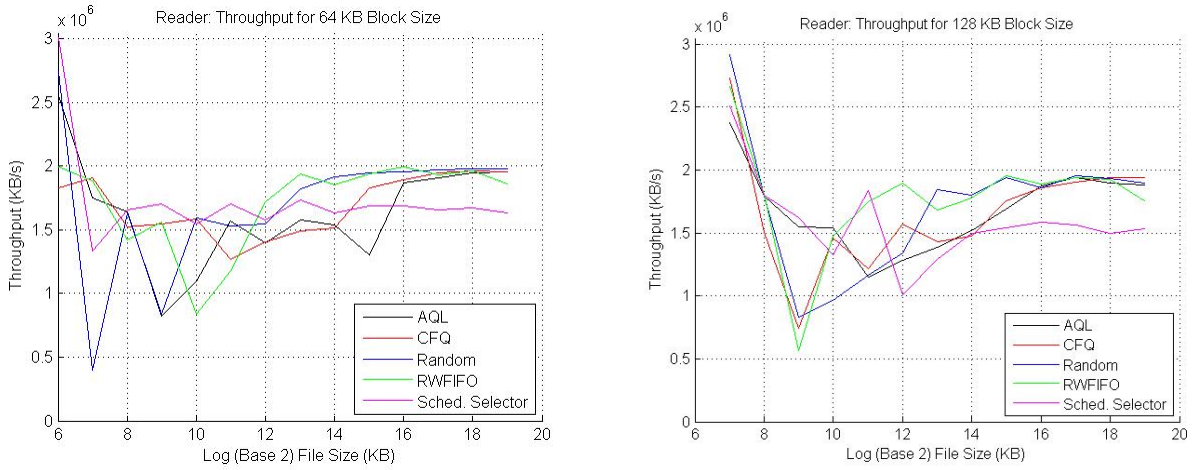

Fig. 1. Throughput measurements for Sequentially Reading Various File sizes in Block sizes of 64k and 128k

VI. Analysis and Observations

This section presents a comparative performance analysis using the workloads described in Section V for the Linux CFQ scheduler, the AQL I/O scheduler, the Read/Write FIFO scheduler, Random I/O scheduler as well as the performance provided by the scheduler selector. The goal of the analysis is to understand how the throughput is affected by the scheduling policy and how different schedulers perform under different workloads.

A. Experiment 1—Multi Process Throughput Measurement

For multiprocess throughput evaluation we let IOzone run 5 processes for the following benchmark tests:
Initial write, Rewrite, Mixed workload, Random write, Read, Re-read, Stride read and Random read. Table I shows the average throughput for each of the five schedulers for different write and mixed workload tests. Table II shows the average throughput for each of the five schedulers for different read tests. The throughputs reported are the overall throughput for the respective schedulers with all 5 processes running. The average throughput per process is substantially lower than the overall value.

The throughput mode test is particularly useful for RWFIFO as it uses a mixed workload with both reads and writes—which is not available for single process throughput measurement in IOzone. From Table I and II it can be seen that RWFIFO has better performance for Reads as compared to Writes and Mixed workload. This is however the case for almost all the schedulers except Random, whose performance degrades for Reads. RWFIFO acts like no-op in the absence of a mixed workload —if there are only reads then it schedules them in a fifo fashion, similarly for only writes case. Under a mixed workload RWFIFO's performance is slightly worse than CFQ's performance. For Reads RWFIFO's performance is comparable to CFQ. The performance of RWFIFO can be explained by the fact that since these are all freshly written copies on the disk. The data blocks are sequentially written and so are the inodes. So, even with no "clever" scheduling, reads can be done quite quickly.

In the random write case (Table I), we see things turning upside down. Because the schedulers usually prioritize read over write, we get lower numbers here for RWFIFO and CFQ. As AQL does not prioritize Reads over writes its performance is better than all the others. Random in this case performs the worst. Amongst the various Read workloads, the worst performance for all the schedulers is for Random Reads (except Random). The reason for the performance decrease for the Random cases is because the blocks to be read/written are not sequential and most of the requests can't be merged,

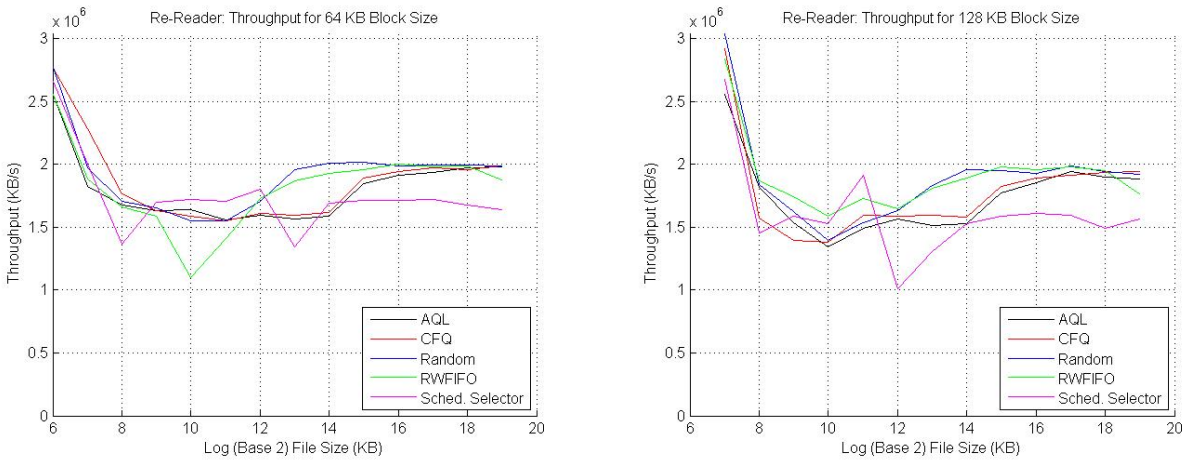

Fig. 2. Throughput measurements for Sequentially Re-Reading Various File sizes in Block sizes of 64k and 128k

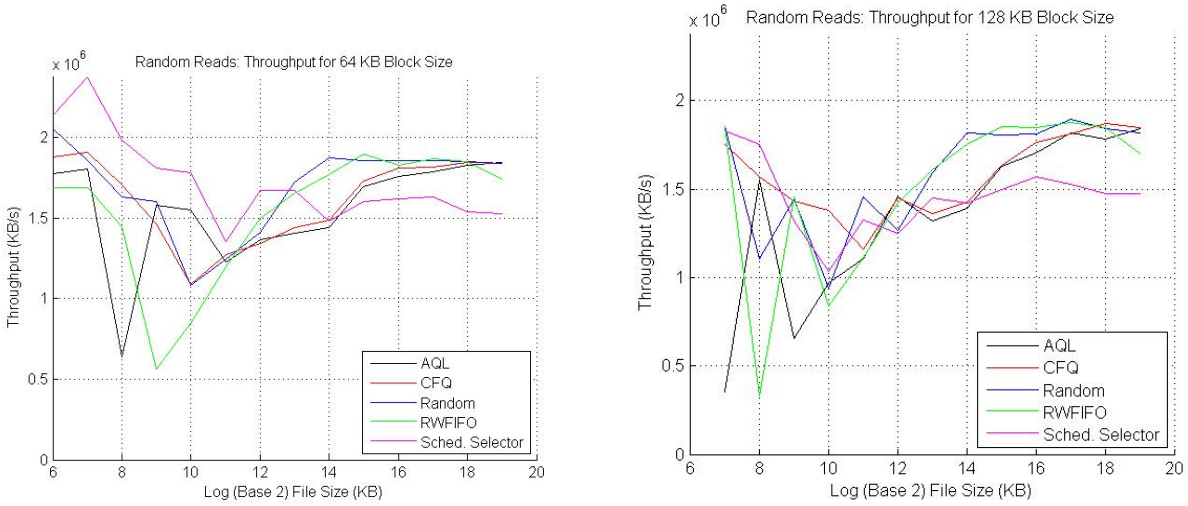

Fig. 3. Throughput measurements for Randomly Reading Various File sizes in Block sizes of 64k and 128k

as a result the disk seek time and rotational time increases as the right block is found on the disk and hence the throughput for the disks decreases. Similarly Stride Read the relative performance as compared to sequential Read is lower—seek time is greater.

From Table I and Table II we can still see that the performance of AQL and CFQ is similar, in fact for Re-write and Random Write AQL has a better throughput than CFQ. The reason for this is explained in Section VI.C.

For multiple processes the variation in the performance of the Scheduler selector (Table I and Table II) for the different workloads is similar to the variations for the other schedulers —it is higher for reads and lower for writes, higher for sequential access, and lower for random and stride access. The performance of the scheduler selector should ideally be as high as the performance of the best scheduler but it will be lower because of the switching overhead to change the schedulers.

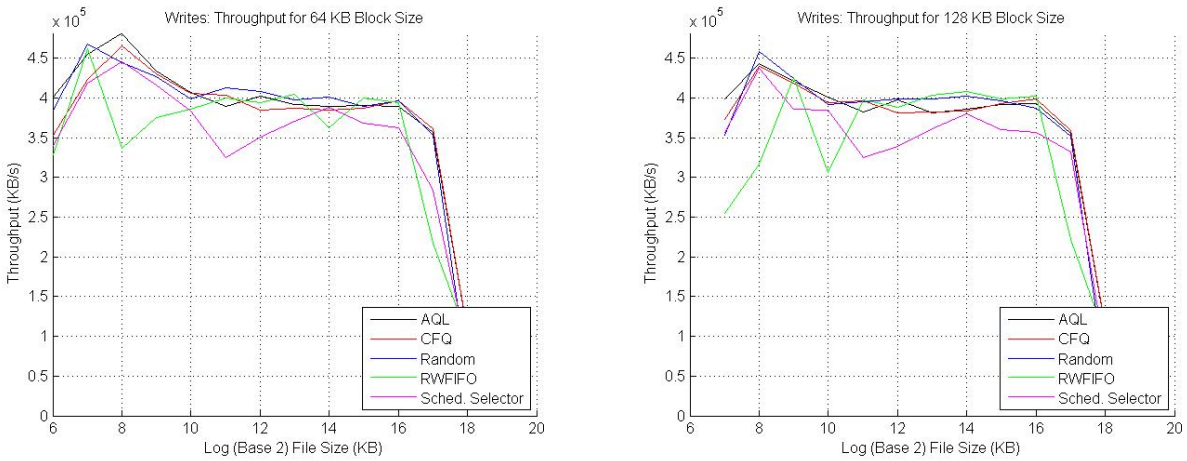

Fig. 4. Throughput measurements for writing to files of different sizes in Block sizes of 64k and 128k including writing file metadata

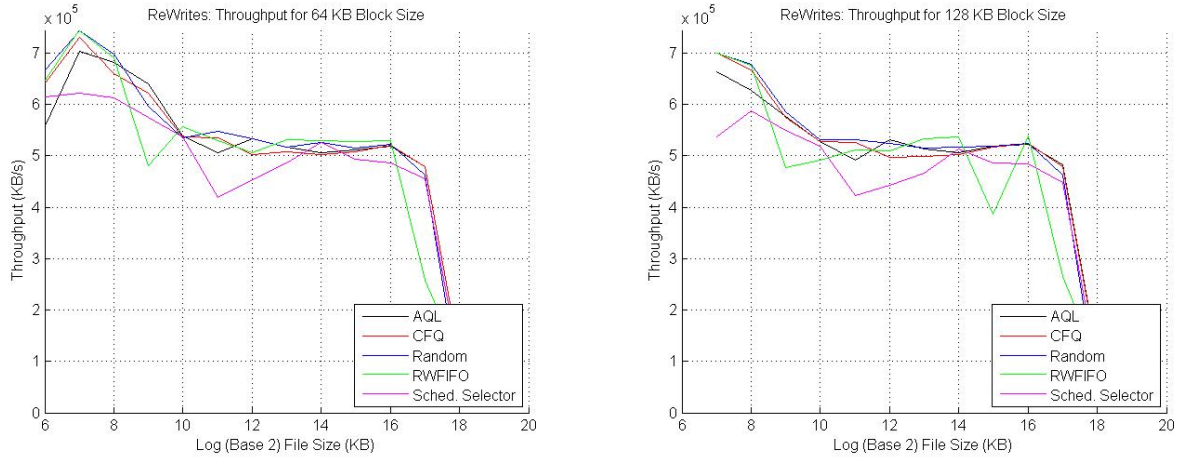

Fig. 5. Throughput measurements for rewriting to existing file of different sizes in Block sizes of 64k and 128k

B. Experiment 2—Single Process Throughput Measurement

We use IOzone in automatic mode, as described in section V to provide throughput measurements for a single process with varying workloads for various file sizes, as described in section V.D using each of the schedulers, AQL, CFQ, Random, RW-FIFO and the scheduler selector. We use the following file sizes 64kB, 128kB, 256kB, 512kB, 1024kB, 2048kB, 4096kB, 8192kB, 16384kB, 32768kB, 65536kB, 131072kB, 262144kB and 524288kB. We use throughput measurements for record sizes of 64kB and 128kB to glean insightful information about I/O scheduler performance. We chose these record sizes because IOzone automatic mode does performance testing for all or nearly all file sizes for those record sizes. We plot the throughput of each scheduler as a function of the log of the file sizes and record sizes and compare their performance. Figures 1 to 6 show the plots. We would also like to note that the data generated was from a single benchmark test for each scheduler and does not reflect the average case for these schedulers. There is also large variations in measured throughput for smaller file sizes because IOzone has no fixed time

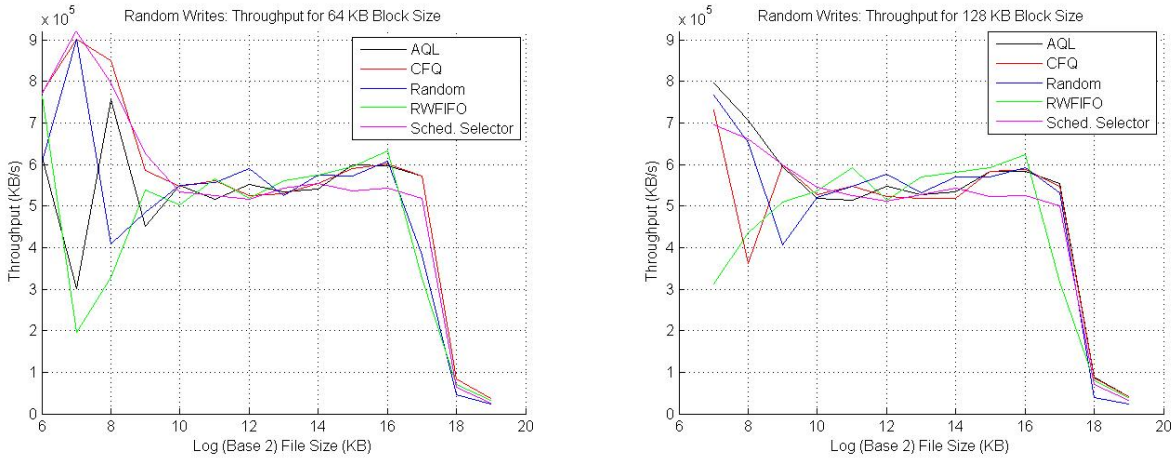

Fig. 6. Throughput measurements for randomly writing to files of different sizes in Block sizes of 64k and 128k

testing.Therefore, our analysis for smaller file sizes may not be relevant or valid.

We see that scheduler behavior does not vary much across record sizes. More or less the same behavior is exhibited by the schedulers for record sizes of 64k and 128k. We also see that the trends exhibited by the schedulers while doing random reads are very similar to the trends exhibited by the schedulers while doing sequential reads. We believe this is due to the fact that all requests are put in the dispatch queue in a sorted manner, so that throughput is not affected by a large amount, though we do see a significant difference in throughput when the file size is small.

The initial writes (Figure 5) performance for all the schedulers is worse compared to rewrites because fresh writes end up writing metadata to disk which is a slow operation. Therefore, it is normal for the writing of a fresh file to have lower throughput than a rewrite due to this overhead. [5] Similarly, initial sequential reads of a file exhibit lower throughput compared to sequential rereads due to memory caching.

In both the random read and random write tests there is no degradation in performance because the file sizes used for testing are not large enough to remove caching as a factor (L2 cache is quite large). We suspect that scheduler performance will begin to degrade once we exceed the cache size.

C. AQL vs CFQ

The results obtained show that CFQ and AQL have similar performance. The reason for this behavior is that in our experiments processes had identical I/O requirements. AQL in this instance, behaves similar to CFQ because the I/O request queue lengths per process will be approximately equal (excluding the case where 2 or more processes may share a queue). But, AQL was designed with varying I/O needs of processes. Therefore, AQL should provide n times I/O bandwidth to a process which is keeping its queue n times longer than another process.

Unfortunately IOzone does not allow for fixed time length throughput testing. Therefore, for future work, we would like to show that AQL differs when processes have differing I/O needs. This would require an experiment where two or more processes are executing I/O operations for equivalent amounts of time and barrier synchronization to ensure they all are executing at the same time. Delay would be introduced in between requests for some of these processes as to simulate varying levels of I/O demand from each process.

VII. Conclusion

The throughput of our schedulers is similar to the performance of that of CFQ. AQL behaves similar to CFQ, while RWFIFO's performance is similar

to no-op and the Random scheduler does better for certain workloads and for certain workloads it performs worse. The scheduler selector did not seem to yield any performance gains. The overall throughput for a scheduler for multiple processes doing IO is lower than the case of a single process.

VIII. Contributions

Joe wrote the AQL and the random I/O schedulers. He wrote the MATLAB scripts to generate the Plots for the analysis section. He wrote Section II.A, Section II.B and Section II.C, Section III.A, Section VI.B, and Section VI.C. He also ran benchmark tests for AQL and CFQ for the automatic mode of IOzone and throughput mode with 5 processes.

Sarah wrote the RWFIFO scheduler. She wrote the abstract, Section I, Introduction of Section II,Section III (other than III.A), Section V, Section VI.A, Section VI.B and the Conclusion. She generated the plots from the MATLAB script and made tables I and II for section VI.A. She also ran the benchmark tests for RWFIFO and Random for automatic mode of IOzone and throughput mode with 5 processes.

Asad wrote the scheduler selector and ran benchmark tests (same as above) for the scheduler selector. He wrote section IV and helped with section VI.B.

References

[1] http://pic.dhe.ibm.com/infocenter/lnxinfo/v3r0m0/index.jsp?topic=%2Fperformance%2Ftune forsybase%2Fioschedulers.htm

[2] https://access.redhat.com/site/documentation/en-US/Red$_{H}at_{E}nterprise_{L}inux/6/html/Performance_{T}uning_{G}uide/ch06s04s03.html$

[3] http://www.linuxinsight.com/files/ols2004/pratt-reprint.pdf

[4] https://access.redhat.com/site/documentation/en-US/Red$_{H}at_{E}nterprise_{L}inux/6/html/Performance_{T}uning_{G}uide/ch06s04s02.html$

[5] http://www.iozone.org/docs/IOzone$_{m}sword_{9}8.pdf$

[6] Seelam, Seetharami, et al. "Enhancements to Linux I/O Scheduling." Proc. of the Linux Symposium. Vol. 2. 2005.